# UNITED STATES PATENT APPLICATION

FOR

# METHOD AND APPARATUS FOR EXTRACTION

INVENTORS:

**ARVIND KRISHNAMURTHY**

**JASWINDER P. SINGH**

**RANDOLPH WANG**

**XIANG YU**

**PREPARED BY WILSON SONSINI GOODRICH & ROSATI**

## CROSS REFERENCE TO RELATED APPLICATIONS

This application is a continuation-in-part of U.S. Application Serial No. filed August 20, 2001 (Attorney Docket No. (25961-707), which claims the benefit and priority to U.S. Provisional Application No. 60/226,479, filed August 18, 2000 (Attorney Docket No. 25961-701), U.S. Provisional Application No. 60/227,125, filed August 22, 2000 (Attorney Docket No. 25961-702) and U.S. Provisional application No. 60/227,875, filed August 25, 2000 (Attorney Docket No. 25961-703). These applications are herein incorporated by reference.

## BACKGROUND OF THE INVENTION

In this section, we first describe what clips are. We then briefly survey the state-of-art of web clip extraction. We then show why these techniques are inadequate in the face of the wide variety and dynamic nature of web pages.

### Field of the Invention

The present invention pertains to the field of computer software. More specifically, the present invention relates to one or more of the definition, extraction, delivery, and hyper-linking of clips, for example web clips.

## SUMMARY OF THE INVENTION

In this document, we describe a web-clipping approach that meets the requirements summarized in the previous section. These algorithms, which we call the *PageDiff* algorithms, are based on the computation of the shortest edit sequence distance. They take both content and structural information into account. They provide the foundation of a powerful content transformation infrastructure that can support many applications and services.

## BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates one embodiment of a web clip by using a drag and drop graphical user interface.

Fig. 2 illustrates one embodiment of extracting web clips by applying a view to a sequence of evolving pages.

Fig. 3(a) illustrates one embodiment of a view definition.

Fig. 3(b) illustrates one embodiment of a clip extraction.

Fig. 4 illustrates one embodiment of extracting a clip by computing a page difference.

Fig. 5 illustrates one embodiment of extracting a clip by applying FlatDiff.

Fig. 6 illustrates one embodiment of permutation of page elements.

Fig. 7 illustrates one embodiment of extracting a clip by applying TreeDiff.

Fig. 8 illustrates one embodiment of a subtree pruning using FlatDiff to improve TreeDiff performance.

Fig. 9 illustrates one embodiment of structural changes that may defeat tree traversal-based extraction.

Fig. 10 illustrates components of the "meta-web" graph.

Fig. 11 illusatrates a TreeDiff using backing-up.

**Web Clips**

A clip is simply a portion or selection of data of an existing document or set of data. The content of a clip may be contiguous or noncontiguous in the source representation of the document or in a visually or otherwise rendered representation. The particular example that we will use in this application is that of web clips, which are portions of existing web pages, though the methods described are application to many other types of documents or sets of data as well. (A document may be thought to contain a set of data, and a clip is a selection or subset of the data.)

In particular, some of the detailed characteristics of the algorithms we describe are tailored to documents containing a markup language, which means that in addition to content (typically text-based content), they also have tags (typically text-based) associated with the content. Examples of markup languages include flavors of HTML (HyperText Markup Language), WML (Wireless Markup Language), various subsets or flavors of SGML (Standard Generalized Markup Language) and various subsets of XML (Extended Markup Language). The tags may specify structural, semantic, or formatting characteristics associated with the document or with specific pieces of content in it. For example, a tag may specify the beginning of a paragraph or table (structure), it may specify that a given number in the document is a product price (semantics), or it may specify that a given text string is to be rendered in bold font and red color (formatting).

The content and markup languages that our methods are specialized toward are typically text-based, e.g. the content and tags are represented in textual form, though the text includes numbers, foreign languages, special characters (e.g. new lines, tabs, other control characters) and the like. The term text-based also includes documents that embed images, video or audio, since the embedding is often specified in text form in the source document (e.g. how a video clip or image is included in an HTML document is specified in text in the source HTML document).

However, the methods described here can also be used for other applications. One example is extracting clips from plain-text documents, i.e. documents that do not have any markup language or tags (the methods are also highly applicable to documents that have only markup language tags and no content). Another is extracting clips from computer programs, which usually have a hierarchical structure just like documents that contain a

markup language often do. The methods described here are applicable to extracting clips from computer programs (for example) as well, though the specifics or the weights given to different entities within the document (e.g. within the source code or some other representation of the computer program) may vary. The methods can also be used for extracting clips (nodes or subtrees) from any tree data structures or information that can be represented as a tree data structure, whether or not those trees represent documents that include a markup language or are derived from documents that include a markup language.

Figure 1 shows an example web clip. Henceforth, we shall refer to web clips for concreteness, rather than to clips in general. A web clip may consist of information or of interfaces to underlying applications or to any other document content.

Figure 1 defining a web clip. The user uses a drag-and-drop graphical user interface to define a "CNN cover story web clip".

Web clips have many uses. One important use is delivering content to the emerging internet-enabled wireless devices. Most existing web pages are authored for consumption on desktop computers where users typically enjoy generous display and networking capabilities. Most wireless devices, on the other hand, are characterized by limitations of small screen real estate and poor network connectivity. Browsing an existing web page as a whole on such a device is both cumbersome (in terms of navigating through the page) and wasteful (in terms of demand on network connectivity). Web clipping can eliminate these inconveniences enabling easy access to any desired content.

We note that web clipping is a complementary but orthogonal technique to other wireless web solutions such as transcoding. In its simplest form, the fundamental problem addressed by web clipping is information *granularity*. The default information granularity on the web is in units of pages. "Transcoders", which are programs that automatically transform existing web pages for consumption on wireless devices using techniques such as reducing the resolution of images, address the information *format* but they do not alter the granularity. As a result, end devices are still flooded with information that overwhelms their capabilities. In practice, one should combine these techniques so that end devices receive content in both the right granularity and the right format.

Web clips are also useful for delivery to portals on personal computers or handheld or mobile devices. Even on personal or desktop computers, portals usually aggregate

content and application interfaces from a multiple sources. Web clips, with or without transcoding, can be delivered to portals or portal software as well. Other example of the use of web clips is in exposing them to users, whether human users or applications, in a remotely or programmatically accessible manner, delivering them to databases or other channels or repositories, converting them to a representation with explicitly identified fine-grained structure even within a clip (such as the Extensible Markup Language or XML) and making them available to devices, transformation systems, applications (that can interact with these structured representations), databases and other channels. Many of these scenarios may require syntactic or semantic transformations to be performed on the web clips---for example, conversion from one description or markup language to another, or format and semantic alterations---but are orthogonal to the extraction of clips from the underlying documents.

**Existing Web Clip Extraction Techniques and Their Inadequacies**

Recognizing the important uses of web clipping, several techniques to extract web clips from pages have been developed, including in a commercial context. In this section, we briefly survey these attempts and their limitations.

**Static Clips vs. Dynamic Clips**

When a user or another entity such as a computer program defines a web clip, which we also refer to as selecting a web clip, the definition is based on a particular version of the underlying page. For example, in Figure 1, the cover story clip definition is based on the CNN page as of June 8th, 2000 at 2:40am. Pages, however, can evolve, in at least three dimensions: content, structure, and name (e.g. URL). In this simple example, the cover story of the CNN home page updates often, and this is the simplest form of page evolution: content change. In other examples, some aspects of the structure of the page (as encoded in its structural and formatting markup language tags and the relative placement of the pieces of data in the page, and to an extent reflected in its layout as viewed for example through a browser that renders the content based on the markup language) may change. Or pages with new names but similar structure to existing pages may be added all the time, e.g. new pages in a content catalog or new news stories (how to deal with changes in name or with pages with new names will be discussed in elsewhere; in particular, the question of which view to use as the original view when a page with a new

name is encountered for extraction; for now, we assume that view to be is to be used and/or the page(s) on which it is defined is known). A challenging question that any web clip extraction technique must address is how to respond to these changes.

A simple solution to deal with changes is not to deal with them at all: the clip "freezes" at the time of clip definition. We call such clips *static clips*.

A different approach is to produce or extract clips that evolve along with the underlying pages. We call such clips *dynamic clips*. In this case, a clip definition or selection specifies which portion of the underlying page is to be clipped. We call such a definition a *view*. The example in Figure 1, efines a "CNN cover story view", and Figure 2 continues the example as we extract different cover stories from the evolving underlying page. The challenge now is to identify which portion of a current page best corresponds to (or has the greatest strength of correspondence with) the portion (or selected set of data) specified in the original view. Determining or identifying this corresponding set of data (or desired clip), is the central problem solved by the technologies described in this document, together with the problem of selecting the most appropriate original view in some cases as discussed later. We refer to the set of technologies as addressing the web clip extraction problem,

**Clip Extraction Based on Characteristic Features**

One approach to the problem of dynamic clip extraction is to identify relatively stable characteristic features either in the clip itself or in the surrounding area of the desired clip. These characteristic features, along with the positional relationship between these features and the desired clip, are stored. Given a new page, the system searches for these characteristic features and use the positional hints to locate the desired clip in the new page. This is often referred to as a rule-based approach.

The disadvantages of this approach are 1) it is labor-intensive, and 2) it is not robust. This is not a general solution that can be automated for any web page; instead, ad hoc solutions must be tailor made for different pages, as different characteristic features must be identified with human aid. It is also an iterative process based on trial and error, as multiple features may need to be tried out before a usable one is identified. It is a fragile solution, as the characteristic features and the positional information may evolve over time as well. Indeed, due to these disadvantages, it is necessary to have a human

"expert" involved in the clip definition process, an expensive and slow proposition that precludes simple do-it-yourself deployment over the Internet.

**Clip Extraction Based on Syntax Tree Traversal**

Instead of relying exclusively on the use of characteristic features, an alternative solution is to exploit the fact that even though the content of an underlying page evolves, its syntactic structure may remain the same. Under this approach, an abstract syntax tree (AST) is built for the original underlying page (for example, based on the structure expressed by the markup language contained in the page), the tree nodes corresponding to the desired clip are identified, and the path(s) leading to a selected node(s) in the original page is recorded. Given a new page that shares the same syntax tree structure, one simply traverses the AST of the new page by following the recorded path and locates the nodes that represent the desired clip.

This solution does not require ad hoc heuristics for different pages. The amount of user involvement required is minimal, so this solution is suitable for do-it-yourself deployment over the Internet. The main disadvantage of this approach is that it relies on the stability of the syntactic structure of underlying page; as the AST of a page evolves, the traversal path leading to the desired nodes changes as well and locating the desired nodes becomes non-trivial.

**Desirable Features of a Good Clip Extraction Algorithm**

In this section, we have briefly surveyed the related efforts in web clip extraction. As a result of analyzing their weaknesses, we can identify a list of desirable features of a good extraction algorithm:

- *Ease of use.* Algorithms should allow views to be specified in very simple ways, e.g. simply pointing and clicking on the desired clips (in the original page) themselves, rather than requiring the specification of complex rules or heuristics by the user.
- *Lack of restrictions in clipping.* Techniques should allow as broad a set of data as possible to be included in a clip, instead of limiting the types of data that can be included in a clip definition or view to, for example, just images, hyper-links, or tables.

- *Freshness of content.* Techniques should be able to extract dynamic clips rather than only static clips.
- *Graceful toleration of changes in page structure.* The need for users to have to redefine view as a page changes in structure or content should be minimized.
- *Graceful handling of URL changes.* When the URL of the underlying page changes, users should not have to be required to explicitly name the view that is to be applied to this URL. Rather, the method should automatically select the most appropriate view as far as possible.
- *Robustness.* Since the underlying page can experience an arbitrary degree of change, we recognize that no clip extraction algorithm can boast 100% success. The goal is to be able to tolerate the greatest amount of content, structural, and naming (i.e. the name of the document or set of data from which the clip is extracted or on which it is defined) changes.
- *Extensibility.* While the system should be easy to use, even by casual users, it may also provide clip-processing infrastructure that can accommodate the more sophisticated transformation needs of power users.

**Detailed Description**

In this section, we first give an overview of the web-clipping process and infrastructure. We then describe in detail a number of web-clipping algorithms and how they can be integrated to provide good performance and robustness. We close by enumerating a number of extensions and applications of the basic web-clipping concept.

**Overview: View Definition and Clip Extraction**

gives an overview of the components and data flow channels involved in clip extraction. Figure 3(a) illustrates how clips are defined. A clip definition is called a *view*. A proxy (named the *view repository* in the figure) retrieves a conventional web page (named *page1*). The proxy then augments this page with some graphical user interface (*GUI*) code and sends the augmented page (named *page1'*) to the user (named *view client*). The view client runs the GUI code to specify the data to be included in the selected clip (which selection of data will later be used for finding the best corresponding data or clip in another page). For example, the user may simply visually point to and select the data that

are to be included in the selected clip, and might not specify anything else. The result (named *view*) is stored in the view repository for later use. It is not necessary that a human user perform the selection of the view; a program or any software can do it as well, in which case GUI code may or may not be needed.

Figure 3(b) illustrates how clips are extracted. The *extraction engine* accesses (obtains proactively or receives) a conventional web page (named *page2*). It then accesses (obtains proactively or receives) an applicable view from the view repository. It calculates which portion of page2 corresponds to the clip specified in the view based on page1. The result (*clip)* is sent to the display device (named *display client*). The figure shows the source of the web page, the machine (computer) that runs the extraction engine, the view repository and the display client being different machines connected by a network. However, it is possible that two or more of these entities (the source of the web page, the machine that runs the extraction engine, the view repository and the display client) be the same machine or run on the same machine. For example, the extraction engine can run on the web server that provides *page2* above. Or the extraction engine can run on the client (for example, when the client is a PC and the clips are being served up into a portal environment). Or all four (the extraction engine, the web server that serves up *page2*, the view repository and the client) can run on the same machine. Any combinations of the different pieces of software and storage running on the same machine or on network-connected machines is possible, though scenarios in which some of the processes run on different network-connected machines are most likely.

The algorithm(s) employed in the extraction engine is one of the key technologies, and it is this component that we describe in detail in the next few subsections. The choice of an applicable view from a view repository is another key technology that is described elsewhere. Other sections describe material supporting three other key technologies related to extraction: namely the adaptation of the view definition itself over time, the repeated application of the extraction algorithm(s) at successively smaller granularities to extract very small clips, and the use of the extraction algorithm to compute a quality measure for the view definition itself in order to give feedback about the quality of definition to the view definer.

**PageDiff: Calculating Page Difference for Clip Extraction**

Although pages evolve, in the vast majority of the cases, the changes are gradual and there is a great deal of commonality in terms of both content and structure between the new page and the old page, upon which the view was originally defined. A successful clip extraction algorithm must be able to exploit commonality as well as tolerate differences as pages evolve.

A key insight behind our technology is the realization that the problem of clip extraction is an extension of the more general problem of finding the shortest edit sequence between two documents. An edit sequence is a sequence of insert, delete, and replace operations that can transform a source document into a target document. The shortest edit sequence is usually called the *difference* between these two documents. For example, the "diff" utility on the Unix operating system uses the shortest edit sequence to compute the difference between two text files. Our approach is to use edit sequences not to find all differences between two files or documents, but rather to find the clip in the new document that best corresponds to the selected portion(s) in the first document. We call this approach of using difference computation to extract web clips *PageDiff.*

Figure 4 illustrates the PageDiff insight. In this figure, two documents are the inputs to the system; one is the web page upon which the view is originally defined; and the other is the new version of the page (or a related or similar page). In one approach to PageDiff, by calculating the difference between these two pages, PageDiff attempts to "match" each piece of one document with a piece in the second document. In this example, the edit sequence contains replacing section B of the first page with section XY in the new page. Since section B is the clip in the old page, the algorithm is in effect declaring section XY to be the clip in the new page because it is the best match for B. As we shall see, it is not necessary to actually compute all the differences between the two pages---as existing edit sequence based methods do---in order to find the clip in the new page that best corresponds to a clip in the old page.

PageDiff is a much more general, elegant, powerful, robust, and flexible approach to the web clip extraction problem than existing approaches. It is general and elegant in that it does not require ad hoc customized solutions for different pages. It is powerful and robust in that it takes *all* commonality between two pages into consideration, in terms of both content and structure; and it can tolerate both content and structural evolutions. Edit

sequence computation is a well-defined and well-studied problem and the theoretical foundation of the solutions further lends a degree of confidence in the robustness of this approach. It is flexible in that depending on the amount of the difference between the two documents, one can choose implementation options of varying computation cost; and there are numerous ways of extending the base implementations.

Tracking changes in pages by computing page differences is not a new idea. The focus of these systems is to allow users to easily identify the changes without having to resort to cumbersome visual inspection, or to reduce the consumption of network bandwidth by only transmitting the page difference to reconstruct the new page on a bandwidth-starved client. However, the insight that one can also adapt page difference computation for the purpose of web clip extraction, and the specific algorithms and improvements for correctness and performance, are the key contributions of this work.

We next describe two implementations of PageDiff: *FlatDiff*, and *TreeDiff*. We then describe how to combine these implementations and existing approaches to achieve good performance, correctness and robustness. In the cases where the defined clip is composed of non-contiguous portions of the first document, the extraction algorithm can be run once to extract all sub-clips or can be run once per sub-clip.

**FlatDiff: Computing Page Difference Between Unstructured Documents**

Algorithms for computing edit sequence to determine the differences between documents are well known. The first approach we present is to treat a web page as a sequence of unstructured tokens and to compute page difference by extending an existing well-known edit sequence computation algorithm. Figure 5 illustrates the process. Some of our extensions to the algorithm are discussed.

A parser first transforms a web page into an abstract syntax tree (AST). The tree is then linearized into a sequence of tokens, which consist of markup elements (defined by the markup language syntax and denoting structure, semantics, formatting or other information as discussed earlier) and text strings that represent the content. The token sequences corresponding to the view and the new page are then fed into the FlatDiff stage, which computes a shortest edit sequence using our extensions to a well-known flat edit sequence calculation algorithm. By locating the matching tokens in the original ASTs, the extraction stage outputs the desired clip.

**Defining FlatDiff**

The key component in Figure 5 is the FlatDiff stage. It attempts to minimize the following edit sequence distance function:

$$C(i,j) = \min[C(i-1,j) + C_d(i), C(i,j-1) + C_i(i), C(i-1,j-1) + C_r(i,j)]$$

where $C(i,j)$ is the edit sequence distance of transforming the first $i$ tokens of document one into the first $j$ tokens of document two, $C_d(i)$ is the cost of deleting the $i^{th}$ token from document one, $C_i(i)$ is the cost of inserting the $i^{th}$ token into document one, and $C_r(i,j)$ is the cost of replacing the $i^{th}$ token of document one with the $j^{th}$ token of document two. $C_r(i,j) = 0$ if the $i^{th}$ token of document one and the $j^{th}$ token of document two are identical.

**Tuning Cost Functions of FlatDiff**

As mentioned earlier, the use of edit sequence to compute all the differences between documents is well known; however, in the context of using FlatDiff itself for our purpose of identifying corresponding clips, there are some enhancements we make to the basic edit sequence calculation algorithm. For example, how to define the cost functions ($C_d$, $C_i$, and $C_r$) is a key question that we must address in order to make FlatDiff work well for the purpose of matching web page elements rather than simply for identifying all differences between pages. One useful observation is that the cost functions can be dependent on the level in the syntax trees (of the document) where the tokens reside: for example, token matches at higher levels of the syntax tree may be deemed to be more significant. This is because these high level matches tend to correspond to matches of high level skeletal page structures. By favoring these matches, we can avoid 'false' matches either across different levels or at lower levels.

We can also manipulate cost functions to deal with more challenging page change scenarios. Figure 6 shows an example involving permutation of page elements: the block denoted by B1 is the desired clip in the old page; its content is changed to B2 in the new page; and its position is changed as well. Since the remainder block, block A, has more tokens in it, a straightforward difference computation algorithm may correctly match block A and produce an edit sequence of [delete B1, insert B2]. In general, however, such an

edit sequence is unsuitable for clip extraction because it is not obvious to the extraction stage that B1 and B2 correspond to each other. Since the goal here is to identify a clip in the new page corresponding to a selected clip in the old page, a solution to this problem is to attach more significance to matches of tokens that are present in the selected clip than to tokens that are not present in the selected clip. In this example, even though blocks B1 and B2 have fewer token in them, their internal matches are given more importance so that the FlatDiff stage will match B1 and B2 and produce an edit sequence of [delete A, insert A]. The extraction stage can readily use this result to produce the correct result clip B2.

**FlatDiff Summary**

In addition to the general benefits of PageDiff, FlatDiff can be implemented efficiently: its complexity is $O(N^2)$ where $N$ is the number of tokens. One of its disadvantages is that it is not always easy to infer the desired clip simply from the edit sequence result. This is because FlatDiff works on unstructured token sequences and the resulting edit sequence also consists of a set of unstructured tokens. It is sometimes difficult to identify syntactically coherent clips using the unstructured FlatDiff output.

**TreeDiff: Computing Page Difference Between Structured Documents**

Unlike FlatDiff, which largely discards structural information in the difference computation step, our second approach, TreeDiff, maintains the structural information throughout the entire process. TreeDiff is a tree-based edit sequence calculation approach, and is directly applicable to documents whose structure can be represented as a tree (examples include web pages, documents containing other formatting or markup languages, computer programs, and many other types of documents), i.e. where a tree representation of the data exists or can be created. It is also applicable to many tree-based data structures in general. Algorithms to compute the edit sequence distances between trees are known. In developing TreeDiff for the purpose of identifying and extracting corresponding clips rather than simply for computing differences between trees, we substantially extend---and combine with new techniques described later---a known edit sequence distance algorithm for unordered trees. Figure 7 illustrates the process with a TreeDiff algorithm, with or without the extensions and modifications. The extensions and modifications we make are discussed in later subsections.

A parser first transforms a web page (or other applicable document) into an abstract syntax tree. The trees corresponding to the view and the new page are then fed into the TreeDiff stage, which computes a shortest edit sequence. By locating the matching subtrees in the original trees, the extraction stage outputs a subtree(s) that corresponds to the desired clip. When subtrees correspond to structural units in the page or its markup language, as they are expected to do, structural integrity of clips is maintained.

**Defining TreeDiff**

The key component in Figure 7 is the TreeDiff stage. It attempts to minimize the edit sequence distance between two trees, where an edit sequence consists of a number of deletions, insertions, and replacements of tree nodes, and these operations are defined as the following:

- *Deletion*: all children of the deleted node become children of its parent.
- *Insertion*: the children of the inserted node consist of its own children as well as children of its new parent.
- *Replacement*: The children of the replaced node become children of the replacing node.
- For nodes that are not deleted, inserted, or replaced, TreeDiff preserves the ancestor-descendent relationship.

Similar to FlatDiff, TreeDiff also associates costs to each of these operations. A simple metric that suffices for many extraction scenarios is outlined below:

- *Deletion* costs 2 units per instance.
- *Insertion* costs 2 units per instance.
- *Replacement* costs 3 units per instance (so that replacement is less expensive than a deletion followed by an insertion).

Other metrics or costs for specific operations, specific types of nodes, or specific nodes may be appropriate for specific applications. As with FlatDiff, the specific costs assigned to operations may vary with the application of the method (for example, the document types or specific circumstances), and in some cases new operations may be defined as well. Also, as with FlatDiff, greater weight may be given to operations on or

matches in some nodes or tokens (or types of nodes or tokens) than others, for example to nodes that are higher in the tree and to matches of nodes that are present in the selected clip from the old page.

Additionally, to improve the accuracy of our difference algorithm (TreeDiff or FlatDiff) for the specific case of web pages written in HTML (HyperText Markup Language), we may use some or all of the following cost metrics or rules that exploit the semantics of HTML components. HTML nodes are broadly classified into two categories: text-based content nodes, which represent the page content to be displayed, and tag nodes, which contain structural, semantic or formatting information associated with the content or text nodes. In addition, tag nodes might have optional attributes associated with them. Given these two categories, our cost metric could be enhanced to express which edit operations are likely to occur in practice. In particular,

- If a text node from the old page is faithfully preserved in the new page, we may associate a negative cost to including such a match in the edit sequence. The negative cost provides an incentive to our algorithm to identify more of those exact matches.
- If a text node from the old page does not appear in the new page, but there exists a string of approximately the same length, we may associate a small positive cost with the operation of replacing the old string with the new string. If the string is to be matched with another string of a substantially different length, a large positive cost may be associated with such a replacement.
- A high positive cost may be associated with the edit operation of replacing a text node by a tag node or vice versa.
- If a tag node from the old page is preserved in the new page with all its attributes intact, we may associate a negative cost with including such a match in the edit sequence.
- If a tag node from the old page appears in the new page, but if the attributes associated with the node have changed, a small positive cost may be associated with the act of matching these two nodes.

These cost metrics and rules or relationships are applicable to FlatDiff as well as to Treediff. These cost metrics were derived after extensive experimentation and an in-depth

study of the nature of changes that are made to web pages. Our algorithms, when deployed with these cost metrics, reliably identify the correspondence between nodes in dynamically changing HTML trees. Similar approaches, with similar or different specific scenarios and cost assignment considerations, will be appropriate for documents containing other markup languages, such as the Wireless Markup Language or WML, various flavors of Extensible Markup Language or XML, or various programming languages or program intermediate form representations. The specific cost assignments and rules may be altered for HTML documents as well.

**Improving TreeDiff Performance by Pruning Subtrees Using FlatDiff**

TreeDiff can be computationally expensive as it computes the edit sequence distance between all possible subtrees of the first tree and all possible subtrees of the second tree. Such an implementation may be too slow to be used for real time clip extraction from documents of significant size. We now describe a key optimization to reduce cost (illustrated by the example in Figure 8. Another key optimization is described in a following section.

In this example, to compute the edit sequence between two subtrees (or trees) we first linearize the two subtrees into two token sequences. We then perform on these two token sequences a *2-way FlatDiff*: computing the difference in the forward direction and then computing the difference again in the backward direction. The 2-way FlatDiff prunes one of the subtrees to isolate a "relevant" sub-subtree. We identify this sub-token sequence by locating the boundary point in each direction beyond which the FlatDiff edit sequence distance starts to increase monotonically. We then feed this pruned sub-subtree into the vanilla TreeDiff in place of the un-pruned subtree. We combine the result of the vanilla TreeDiff with the result of the FlatDiff to form the final answer. As a result of this optimization, the size of the subtrees participating in an invocation of the vanilla TreeDiff method is significantly smaller.

**Improving TreeDiff Performance: Subtree Matching**

In the algorithms presented so far, the input includes the AST corresponding to the old HTML page (T1), a distinguished node(s) n1 inside T1, and a new tree, T2, corresponding

to the new HTML page. The difference algorithms compute the mapping from the nodes in T1 to the nodes in T2. Given such a mapping, we can identify whether the distinguished node from the old page is preserved in some form inside the new page, and if so, the subtree rooted at that node in T2 is the clip to be extracted. A variation of this algorithm is obtained by rephrasing the mapping problem in the following manner: given the subtree rooted at n1, what is the subtree in T2 that has the smallest edit distance to the given subtree? At first sight, since rephrasing the question in this form requires the algorithm to consider every subtree in T2 and compute the edit distance of that subtree from the subtree rooted at n1, it would appear that answering this question would require a substantial amount of computation. However, an integral part of the TreeDiff algorithm is to compute the edit distances between **every pair** of subtrees in the old and new ASTs. In fact, given a subtree S1 from T1 and a subtree S2 from T2, the edit distances between these two subtrees is computed by extending the partial results obtained from computing the edit distances for every pair of subtrees inside S1 and S2. Given such an algorithmic behavior, the reformulation does not require further computational enhancements to our original TreeDiff algorithm. In fact, since nodes in T1 that are not descendents of n1 need not be considered to answer the posed question, the input to the algorithm is pruned, resulting in a more efficient algorithm.

However, such pruning results in the loss of contextual information regarding the structure and content of the tree around n1. This loss of information could result in scenarios where our algorithm would identify matches that are "spurious" when the trees are considered in their entirety. To overcome this problem, we introduce the strategy of "back-off", where progressively larger trees inside T1 are considered if the node(s) inside T2 that matches n1 is ambiguous. It is easy to identify when a match is ambiguous: when there are two or more subtrees inside T2 that have similar edit distances from the subtree rooted at n1, we could declare that the algorithm couldn't identify a strong enough or unique enough correspondence or match. When such a situation arises, we consider the subtree rooted at the parent of n1, and identify what subtrees in T2 are similar to this subtree. By including the parent of n1 and the subtrees rooted at the siblings of n1, we are increasing the contextual information used for identifying matches. If this too is inadequate, we back-off one level higher in T1. Once the best larger matching subtree is found, the best clip corresponding to the view can be obtained from within it. Thus, if

unambiguous matches are indeed available in T2, this strategy should eventually result in finding them.

While back-off strategies are well suited to tree-based algorithms due to the inherent hierarchy in the representation, they can be used with flat diff approaches as well, by backing off to subsequently higher-level structural elements each time (e.g. paragraphs, sections, etc.) and performing flat diffs with those larger sub-documents.

**TreeDiff Summary**

TreeDiff preserves the structural information throughout the entire process. One of the consequences is that the matches found by the algorithm are always structurally coherent. This makes the extraction stage simple to implement. The disadvantage of TreeDiff is its relatively high computational cost: TreeDiff has a complexity of $O(N^2 \cdot D^2)$ where N is the number of tree nodes and D is the depth of the tree.

**Integrating Clip Extraction Technologies**

So far, we have described three clip extraction technologies that all take advantage of the syntactic structure of the web page at some stage of the algorithm:

- *Tree traversal*: it has a complexity of $O(D)$; but it cannot tolerate structural changes.
- *FlatDiff*: it has a complexity of $O(N^2)$; it addresses both content and structural changes; but the structural integrity is not maintained.
- *TreeDiff*: it has a complexity of $O(N^2 \cdot D^2)$; it addresses both content and structural changes; and it maintains structural integrity.

In this section, we describe various ways of combining these algorithms.

**Hybrid Integration**

*Hybrid integration* refers to modifying one of these algorithms by incorporating elements of other algorithms. The optimization technique of augmenting TreeDiff with FlatDiff is an example of hybrid integration. We now discuss how to augment tree traversal.

A vanilla tree traversal approach cannot tolerate structural changes that affect the traversal path to the desired node. Figure 9 shows an example: the addition of node I in Page2

interferes with locating the next node on the path, namely node C. It is possible to augment tree traversal with localized FlatDiff or TreeDiff. As we traverse down a path, if we detect structural changes that are likely to defeat tree traversal by, for example, noticing changes in the number of children at the current tree level, we may invoke difference computation of the two subtrees rooted at the current node. In the example of Figure 9, we compute the difference between the two shaded subtrees (rooted at node D). The difference computation matches the components of the subtrees and allows the tree traversal to recover (at Node C).

The advantage of the hybrid integration is obvious: for components of the path that have not changed, tree traversal progresses rapidly; and the more computationally intensive algorithm is only invoked on localized subtrees that hopefully contain a much smaller number of nodes.

Another approach to integrate the various strategies is to reformulate the clip extraction problem to develop a metric that considers structural similarity between the source and target clips as well as the similarity of the paths used to traverse the trees in reaching the clips. We use a cost metric, which given a source clip and a potential target clip, associates a value that is the weighted sum of the TreeDiff edit distance between the two clips and the FlatDiff edit distance between the traversal paths to the clips from the roots of the corresponding trees. This hybrid strategy ensures that our extraction algorithm identifies a target clip such that neither the structural nature of the clip nor its position has changed significantly.

**Sequential Integration**

Suppose we notice a structural change that demands difference computation. The choice that we face now is between FlatDiff and TreeDiff. Unlike hybrid integration, which modifies one algorithm by incorporating elements of other algorithms, *sequential integration* employs multiple algorithms in succession if necessary. Under sequential integration, we will attempt FlatDiff first, examine the result, and if the result fails a correctness test, we will resort to TreeDiff.

This approach is based on the simple observation that verifying the validity of the result can be far more efficient than computing the exact result directly: it is possible to

verify in linear time that the result produced by FlatDiff should match that of a full-blown TreeDiff, thus avoiding the latter.

**Integration Summary**

In this section, we have seen that it is possible to combine the various syntax tree-based algorithms, either in a hybrid fashion, or sequentially. The goal is to rely on the faster algorithms most of the time on a majority number of the nodes and only resort to slower algorithms less frequently on a smaller number of nodes. As a result, we can harvest the best performance and robustness that the various algorithms have to offer.

**Adaptation Over Time and Periodic Extraction**

A long time gap between the definition of a view and its application may allow the target page to experience several generations of structural change, the cumulative effect of which may become too complex for the simpler algorithms to succeed. To cope with this challenge, as our system polls the target page of a view periodically, it refreshes the view definition by applying the clip extraction mechanism and storing the fresher version of the page and its clip in place of the old view, allowing the system to adapt to incremental changes smoothly instead of allowing gradual changes to accumulate beyond the system's ability to recognize them using simpler means. The polling of the target page and the updating of the view definition can be done either on-demand, as the view is accessed, or on a scheduled basis (e.g. every 15 minutes).

The idea here is the following. When a target page P2 is accessed for extraction of a clip, it uses a view definition, which includes a page P1 on which a view is defined. Using the algorithms described above, a clip corresponding to the view (defined on P1) is extracted from the target page P2. Let us assume now that P2 is stored, along with the extracted clip being identified within it somehow, just as the original clip was marked in P1 as part of the view definition. The next time a new target page P3 is accessed in order to extract the corresponding clip (i.e. a clip corresponding to that defined on P1), there are choices regarding which page to use as the view definition. One choice is to use the original page P1 on which the user originally defined the view. Another choice is to use the most recently accessed target page corresponding to this view together with the clip that was extracted from it (i.e. P2, which was stored the previous time). Our system enables P2 and it's clip or view definition to be used, thus allowing the definition of a

view to evolve over time. The fact that the view definition evolves with changes to the page or document ensures that the view definition that is used for a clip extraction is not very old but rather is based on a page that is recent and therefore likely to be more similar to the target page. This helps the correctness of the extraction algorithm, as mentioned earlier, and also likely its performance, as the differences among the pages being compared are likely to be smaller than if the original user-defined view were used.

While simply refreshing the view definition as described above is sufficient for some pages, for others, this technique needs to be extended to maintaining a certain amount of page format history. For example, for a site that regularly cycles through several page formats or switches back and forth among them from time to time, keeping a history of view definitions based on these different formats allows our system to perform object extraction efficiently and accurately using the most appropriate definition at a given time.

The updating of the view definition can be done either each time a view is accessed by a user or application for its particular purpose, or on a scheduled basis (e.g. every 15 minutes or in a manner based on the frequency of changes to the page or other environmental factors).

Periodic scheduled extraction has other benefits. First, the fact that recent pages and their recently extracted clips are stored in or near the machine that performs extraction enables them to be reused like a cache. That is, if the page has not changed since the last time it was accessed and extracted from, there may not be a need to fetch the target page from its original server or to perform the extraction process.

Second, periodic or schedule extraction can be used to support monitoring of the extraction or clip delivery system, whether or not the view definition is updated with successive extractions. At each periodic extraction, a determination can be made whether the extracted clip has enough of a 'match' with the defined clip or view that the system is working correctly and delivering the desired clip. If not---for example if the desired clip is no longer in the page at all or if the algorithm is not succeeding in identifying the desired clip or a clip with a strong enough match---a user or administrator can be notified so that they can take corrective action (such as modifying the view definition appropriately).

**Repeated Invocation to Extract Successively Smaller Clips**

So far we have been discussing the extraction of a clip from a page. It is possible to invoke the extraction algorithm(s) repeatedly to extract successively smaller sub-clips from successively smaller clips. The user may define a view, and then within that view define a sub-view or sub-views, and within those define sub-sub-views, and so on. When a new page is obtained, the extraction algorithm can be run once to extract the highest-level clip from the new page; then again to extract the next-level sub-clip(s)---corresponding to the sub-view(s)---from the extracted clip (treating the clip as the whole document for this second invocation); then again to extract the next-level sub-sub-clip(s)---corresponding to the sub-sub-view(s)---from the extracted sub-clip(s) (treating the sub-clip(s) as the whole document(s) for this third invocation); and so on.

There can be several reasons to do. For one thing, the user may want a very small clip from a page, and the extraction algorithm may not be able to extract the corresponding small clip very reliably from a new page since there not be a strong enough unique match (e.g. the 'wizard' may tell the user this). One choice would be for the user to define a larger view, that contains the desired data within it but is more uniquely identifiable within the page. But the user may not want the corresponding larger clip to be extracted and delivered. The desired view is too small to lead to unique or reliable enough extraction uniquely, and the larger view that is reliable enough is undesirable. In such a situation, the user may define the larger view, which leads to reliable extraction of a clip, and then within it define the smaller view---which leads to reliable extraction from within the larger view (not from within the whole document at once). This two-step (possibly extended to multi-step) extraction process may well lead to the small clip being extracted reliably in situations where a one-step extraction does not lead to a strong or unique enough result.

Another important and related use of sub-clip extraction is to give fine-grained structure to the content of clips. For example, if a clip contains stock quotes for a particular stock ticker symbol, the clip is extracted as an undifferentiated 'blob' of content in a markup language (e.g. HTML). The clip does not provide any structured information about the meaning of its content. It may be desirable to give structure to at least some of the content in the clip. For instance, if the different pieces of the content are tagged as 'stock ticker symbol,' 'stock price,' 'percentage change,' 'volume,' 'exchange,' 'input box for ticker entry.' etc., then the tagged fields that result from extraction can be used in

various ways. The user may define formatting or semantic transformations on the extracted data, such as computing how close the stock is to it's 52-week high price, or the user may define alerts on specific numerical data (e.g. alert me when the stock of company X falls below $70, or other applications or systems may use the structured data for programmatic access and manipulation. That is, just like internally undifferentiated clips of content in a markup language can be used effectively for display on mobile devices or in portals, internally structured clips can be used effectively for access and manipulation by other applications.

Sub-clip extraction may be specified and performed as follows. The user may first define a view. Within the view, the user may select certain sub-views or sub-snippets and give them tags as illustrated above. When a new page is to be extracted from, first the clip corresponding to the view is extracted. Then, the clip is treated as the new document and the defined view as the old document, and sub-clips are extracted from it using the sub-view definitions. This leads to reliably extracted and appropriately tagged structured sub-clips being available for manipulation, transformation and reformatting, or programmatic access by other applications.

**Choosing an Appropriate View to Apply to a Page**

So far, we have defined views that can only apply to fixed pages that are identified by fixed URLs. The second generalization allows for *a wild-card view*, a view definition that can apply to multiple pages with different URLs but identical page formats. For example, the URLs embedded in the CNN home page change constantly. A wild-card view defined for the page pointed to by one of these changing URLs is also applicable to all other "similar" pages that exist then or will exist in the future. Given an arbitrary page, the challenge is to identify the wild-card view(s) that may be applicable to it. Our system uses a combination of three approaches to solve this problem:

- The *URL-based* approach compares the URL of the original page that defines the view to the URL of the arbitrary new page. If the two URLs "match", for some definition of a "match", such as a longest prefix match, we declare the view to be applicable to this new page.
- The *AST-based* approach names pages not by their URLs, but by a concatenation of the AST paths, each of which identifies a tree node within a page encountered

during a hypothetical navigation session. So even when URLs change, constant AST navigational paths can be used to identify the applicable view(s).

- The *structure-based* approach examines the syntactic structure of an arbitrary page and calculates a checksum that is used as an identifier for an applicable view.

When we encounter a page on which there have been no clips defined, a structure-based approach would require identifying whether the user has defined views on a page that is structurally similar to the current page. A faithful implementation of this approach would require measuring the edit distance between the structure of the current page with all other pages stored in the view repository and choosing a page that has the minimum edit distance. However, this approach is expensive and unlikely to scale. Hence, the need for a fast algorithm that approximates this computation without significant loss in accuracy.

We therefore employ an approximate algorithm that employs two techniques to perform structural comparisons. First, the structure of every subtree in the AST is mapped to a single checksum value by combining the hashed values of the individual tag-nodes contained in the AST. Second, we consider the checksums only for those subtrees that are within a certain distance from the root of the AST. Using this pruned list of checksum values for two ASTs, we can use the FlatDiff algorithm to compute an approximate measure of how much the two ASTs differ in their structural representation. Observe that the performance optimizations are derived from the use of a computationally less expensive FlatDiff algorithm and the pruning of the set of subtrees that are considered for structural comparison. These algorithmic design choices result in a system that is efficient without sacrificing on accuracy.

**Using the Quality of the Results of Extraction to Provide User Feedback**

The goal of the extraction algorithm is to find the clip (or sub-clip) with the strongest match to or greatest strength of correspondence with the view (or sub-view). As was discussed earlier, it is possible that multiple clips within a page match the view to some extent. The hope is that one match dominates the others, so a unique clip can be extracted with high confidence. For example, achieving a more unambiguous match is part of the goal of the backup method discussed earlier.

However, it is possible that multiple clips provide matches that are close to one another in the strength of correspondence match, as computed by overall edit distance or some other metric. In this case, not enough certainty is achieved by the algorithm regarding the best match to the selected clip(s) in the view. The extent to which a single clip dominates other possible clips in its strength of correspondence to a selected clip in the view, and it has a high enough strength of correspondence itself, may be used to assign a measure of quality to the view definition (or the definition of that clip in the view). If this measure of quality is high enough, for example if it is above a threshold value, that means that a match that is both unambiguous enough and good enough match has been found. If it is not high enough, feedback may be given to the user that this situation has occurred, so that the user may alter the definition of the view such that the data selected within the view are more unique within the page, and hence to hopefully lead to more unique and strong matches in the future. For example, in an extreme case if a user defines as a view only a single number or word within a page, it is likely that the algorithm will not find a unique enough match or a strong enough match based on content and structure or context.

The quality measure associated with the view definition is impacted negatively if the back-off method does not lead to an unambiguous best match, or even if the back-off method is invoked to being with (even if it ultimately leads to an unambiguous best match), and if the strength of the correspondence (match) of the best matching view is low. The reason that the invocation of the back-off method lowers the quality measure is that the need to invoke the back-off method implies that initially (by examining only the selected portion in the first document) a strong enough or unique enough match could not be found, and the back-off method had to be used to find a more unique match. Thus, the extent to which back-off is used, together the final relative strengths of correspondence of clips to the selected data, can be used to determine the quality measure ascribed to the view definition.

In fact, this approach of giving feedback in the case of ambiguous matches may be used to provide a user feedback at view-definition time. As soon as a user defines a view and saves the definition, the extraction algorithm can be run. This extraction may be done from the latest version of the page (which is in many cases likely to not have changed at all from the page on which the view was defined) or from an earlier version that has been

stored before view definition time. If the quality measure ascribed to the view definition, as described above, is not high enough, the user is given feedback that the view is not defined well enough. The user may then alter the definition of the view, for example by including more data around the selected data, in order to make the view definition more unique within the page.

Thus, this method can be used to create a view definition 'wizard' or software assistant that helps the view-defining user (human or software) define better views by providing feedback about this measure of quality of the view definition to the user. The wizard may be run immediately upon view definition, as described above, or it may be scheduled to run one or more times after some periods of time, so that it is likely that the page will have changed and the extraction to test view definition quality will be done from changed pages rather than the very page on which the view was defined, resulting in more realistic and potentially more informative tests.

**Generalizing the Definitions of Views, Clips, and Hyper-Links**

So far, we have presented the concept of view definition and clip extraction in the context of extracting a single clip from its enclosing page. The definitions of views and clips, of course, can be much broader.

First, a view can be a *composite view* that contains a number of *sub views*, each of which specifies a clip from a page in the manner described above. In fact, these sub views need not even belong to the same enclosing page. As a result of applying these composite views, the clip extraction process can build *composite clips* that are made of smaller clips. Indeed, this process is not only useful for delivering finer-grained information to wireless devices, it is also useful for purposes such as aggregating content from different sources and delivering larger grained clips to large displays that, for example, can be as large as wall-sized.

The second generalization of the definition of a view allows it to be an arbitrary piece of code that performs some computation and transformation on source pages (or clips) and produces *derived clips*.

So far, our description is based on user-defined views of various kinds. A third generalization addresses pages that do not have views associated with them. For such pages, our system breaks them down into smaller objects solely based on their syntactic structure. For example, lists, tables, and images are obvious candidates. Compared to clips generated by applying user-defined views, one disadvantage of these automatically generated objects is that they do not necessarily possess user-friendly look and meaning.

The fourth generalization extends the definition of hyper-links and introduces the concept of a *meta-web*. The key is to recognize that as a result of introducing views and clips onto the web, we now have a much richer link graph. Figure 10 illustrates the components. A graph consists of nodes and edges. There are three types of nodes in our system: web pages, views, and web clips. There are two types of edges: a *definition edge* between node A and node B denotes that component A is defined in terms of component B, and a *reference edge* between node A and node B denotes that there is a hyper-link to component B within component A.

More specifically, Figure 10 shows that there are many circumstances under which such edges can occur. The numbers in the following list correspond to the number labels in Figure 10. A view is defined in terms of an enclosing web page.

1. A clip is defined by a view.
2. A composite view can be defined in terms of other views.
3. Web pages reference views.
4. Clips references web pages.
5. Web pages reference clips.
6. Clips reference each other.
7. Clips reference views.

In particular, note that our system has given rise to two new types of hyper-links. One new type is hyper-links to views (items 4 and 8 above). For example, a web page can link to the "current CNN cover story view". The second new type is hyper-links to clips. For example, a web page can link to the "CNN midnight cover story on 07/09/2000". We call this rich graph centered around views the *meta-web*; and the view-based clipping

architecture is merely a first step towards our goal of providing an infrastructure that "liberates" data from the confines of pre-defined pages and places the most relevant information at users' finger tips.

Some more information on Tree-based Extraction

The TreeDiff algorithms that we have described in the earlier sections perform a computation that unfolds in the following manner. In order to compute the edit distance between two trees: a "source tree" and a "destination tree", it requires the edit distance values for every pair of subtrees enclosed within the two trees. These edit distance values are considered partial solutions and are extended using a dynamic programming algorithm to find the edit distance between progressively larger subtrees. The "edit script" for each intermediate step consists of three kinds of edit operations that operate on entire *subtrees* (instead of operating on individual nodes): deletion, insertion, and replacement of subtrees. This increase in the granularity of the edit operations (from individual nodes to subtrees) is a direct result of expressing the algorithm as a dynamic programming computation. While operating with tree-sized edit operations does speed up the computation of each intermediate step, it has the unfortunate consequence of having to decompose the tree-sized edit operations into smaller node-level edit operations once the entire difference computation comes to a halt.

In order to perform this decomposition, there are two alternatives that expose a time-space trade-off. One approach is to store the edit script for every sub-tree comparison performed during the entire algorithm. At the end of the process, one just needs to unfold a tree-sized edit script into a corresponding node-sized edit script by recursively incorporating the previously stored edit scripts between progressively smaller subtrees. The other approach is to discard the edit scripts, but store just the numeric edit distance values, which are the only pieces of information required by further steps of the dynamic programming algorithm. During extraction, when a tree-sized edit operation needs to be decomposed, we could recompute the edit script resulting in an algorithm that performs more computation but uses substantially less space. However, the amount of redundant computation is a small fraction of the overall computational cost due to the following reason. Since we are interested in finding the replacement for a single target node in the source tree, the algorithm needs to decompose only the tree edit scripts that involve replacements of subtrees that enclose the target node in the source tree. Consequently, the number of recalculations that we must perform is at most equal to the depth of the tree.

We now illustrate this process using an example. In Figure 11(a), we show the source tree $S$, the destination tree $T$, and the target node in the source tree $E_1$. Our task is to find the corresponding node in tree $T$. Figure 11(b) shows the result of the first step of a TreeDiff algorithm: node $E_1$ is compared against all possible subtrees of $T$. Both $E_2$ and $E_3$ are determined to be close enough to $E_1$ so the TreeDiff result so far is inconclusive and we must continue. Figure 11(c) shows the result of the second step of the TreeDiff algorithm: after we "back up" one level, the subtree rooted at $C$ is compared against all possible subtrees of $T$. The two subtrees rooted at $C$ in the destination tree $T$ are both determined to be close enough to the corresponding subtree in $S$ so the TreeDiff result so far is inconclusive and we must continue. Figure 11(d) shows the result of the third step of the TreeDiff algorithm: after we "back up" one more level, the entire source tree $S$ is compared against all possible subtrees of $T$. $S$ is deemed to match $T$ and since the match is unique, the TreeDiff algorithm halts.

Now we must extract the target node from the destination tree by taking advantage of the TreeDiff result, which is expressed as the edit script shown in Figure 11(d). Note the operations numbered 3 and 5 are edit operations on entire subtrees instead of individual nodes. From this edit script, we see that the desired target node $E_1$ is part of edit operation 5. To identify the corresponding target node in $T$, we must decompose this operation into node-sized operations. We perform a redundant computation to find the edit distance to accomplish this decomposition and this decomposition is shown in Figure 11(f). Since this last edit script involves only node-sized edit operations, no further decomposition is necessary and we have concluded finally that $E_2$ is the node that we seek.

In this example, we have used one flavor of a TreeDiff algorithm that uses "backing-up". We note that the extraction algorithm is not dependent on the particular flavor of the TreeDiff algorithm and the extraction algorithm is applicable to all flavors of TreeDiff.

Figure 11 TreeDiff with backing up (a-d) and subsequent extraction (d-e). Subtrees that are deemed to match each other are marked. (a) The source tree $S$, the destination tree $T$, and the source target $E_1$. (b) First step of TreeDiff. (c) Second step of TreeDiff after backing up once. (d) Third step of TreeDiff after backing up again. (e) The

edit script that transforms *S* to *T*. (f) The edit script that transforms the subtree containing the target.